Dictionary-Symbolwise Flexible Parsing

Maxime Crochemore[1,4] Laura Giambruno[2], Alessio Langiu[2,4], Filippo Mignosi[3], Antonio Restivo[2]

[1] King's College London, London WC2R 2LS, UK
maxime.crochemore@kcl.ac.uk
[2] Dipartimento di Matematica e Informatica, Università di Palermo, Italy
[lgiambr,alangiu,restivo]@math.unipa.it
[3] Dipartimento di Informatica, Università dell'Aquila, Italy
filippo.mignosi@di.univaq.it
[4] Université Paris-Est, France

Abstract. Linear time optimal parsing algorithms are very rare in the dictionary based branch of the data compression theory. The most recent is the *Flexible Parsing* algorithm of Mathias and Shainalp that works when the dictionary is prefix closed and the encoding of dictionary pointers has a constant cost. We present the *Dictionary-Symbolwise Flexible Parsing* algorithm that is optimal for prefix-closed dictionaries and any symbolwise compressor under some natural hypothesis. In the case of LZ78-alike algorithms with *variable costs* and any, linear as usual, symbolwise compressor it can be implemented in linear time. In the case of LZ77-alike dictionaries and any symbolwise compressor it can be implemented in $O(n \log(n))$ time. We further present some experimental results that show the effectiveness of the dictionary-symbolwise approach.

1 Introduction

In [19] Mathias and Shainalp gave a linear time optimal parsing algorithm in the case of dictionary compression where the dictionary is prefix closed *and* the cost of encoding dictionary pointer is constant. They called their parsing algorithm *Flexible Parsing*. The basic idea of *one-step-lookahead parsing* that is at the base of flexible parsing was firstly used to our best knowledge in [7] in the case of dictionary compression where the dictionary is prefix closed *and* the cost of encoding dictionary pointer is constant *and* the dictionary is static. A first intuition, not fully exploited, that this idea could be successful used in the case of dynamic dictionaries was given in [8] and also in [12], where it was called parsing MTPL (maximum two-phrase-length parsing).

Optimal parsing algorithms are rare and linear time optimal parsing results are rather rare. We can only also cite the fact that greedy parsing is optimal and linear for LZ77 alike dictionaries and constant cost dictionary pointers (*cf.* [22]) and its generalization to suffix closed dictionaries and constant cost dictionary pointers (*cf.* [2]) later used also in [12].

In this paper we consider the case of a free mixture of a dictionary compressor and a symbolwise compressor and we extend, in some sense, the result of Mathias and Shainalp. We have indeed an optimal parsing algorithm in the case of dictionary-symbolwise compression where the dictionary is prefix closed and the cost of encoding dictionary pointer is *variable* and the symbolwise is any classical one that works in linear time. Our algorithm works under the assumption that a special graph that will be described in next section is well defined. Even in the case where this condition is not satisfied it is possible to use the same method to obtain almost optimal parses. In particular, when the dictionary is LZ78-alike our algorithm can be implemented in linear time and when the dictionary is LZ77-alike our algorithm can be implemented in time $O(n \log(n))$.

The study of free mixtures of two compressor is quite involved and it represents a new theoretical challenge. Free mixture have been implicitly or explicitly using for a long time in many fast and effective compressors such as gzip (*cf.* [6]), PkZip (*cf.* [11]) and Rolz algorithms (*cf.* [16]). For a quick look to compression performance on texts see Mahony challenge's page (*cf.* [15]).

In our case, using a simple static Huffman coding as symbolwise compressor we improved the compression ratio of the *Flexible Parsing* of $6\% - 4\%$ on texts such as prefixes of English Wikipedia data base with a negligible slow down in compressing and decompressing time. The slow down comes

from the fact that we have to add to the dictionary compression and decompression time the Huffman coding and decoding time. The same experimental result holds in general when the dictionary is LZ78-alike. Indeed a dictionary-symbolwise compressor when the dictionary is LZ78-alike and the symbolwise is a simple Huffman coding with optimal parsing has a compression ratio that is is more or less 5% better than the compression ratio of a pure pure LZ78-alike dictionary compressor that uses an optimal parsing. In general smaller is the file greater is the gain. The 5% refers to text sizes of around 20 megabytes. Moreover, preliminary results show that using more powerful but still fast symbolwise compressor, such as an arithmetic encoder of order 1, there is a further 10% gain in compression ratio.

When the dictionary is, instead, LZ77-alike the gain in compression when we use a dictionary-symbolwise compressor with optimal parsing and Huffman coding with respect to a pure dictionary compressor with optimal parsing reduces down to more or less 3%. The compression ratio seems to be sensibly better than in the case of LZ78-alike dictionaries when we use, in both cases, unbounded dictionaries. The distance, however, between the compression ratio of dictionary-symbolwise compressors that use LZ78-alike dictionaries and the ones that use LZ77-alike dictionaries is much smaller, following our preliminary results, when we use an arithmetic encoder of order 1 instead than an Huffman encoding.

We have experimental evidence that many of the most relevant commercial compressors use, following our definition, optimal parsing in the dictionary-symbolwise case where the dictionary is LZ77-alike. The method described in this paper therefore has as a consequence the possibility of optimizing the trade-off between some of the main parameters used for evaluating commercial compressors, such as compression ratio, decompression time, compression time and so on.

So, why linear time optimal parsing algorithms are rather rare? Classically (*cf.* [21]), for static dictionaries it is possible to associate to any dictionary algorithm $\mathcal{A}$ and to any text T a weighted graph $G_{\mathcal{A},T}$ such that there is a bijection between optimal parsings and minimal paths in this graph. The extension of this approach to dynamical dictionaries has been firstly studied, to our best knowledge, in [20] and it has also been later used in [5]. More details will be given in next sections. The graph $G_{\mathcal{A},T}$ is a Directed Acyclic Graph and it is possible to find a minimal path in linear time with respect to the size of it (*cf.* [3]). Unfortunately the size of the graph can be quadratic in the size of the text and this approach was not recommended in [21], because it is too time consuming. From a philosophical point of view, the graph $G_{\mathcal{A},T}$ represents a mathematical modeling of the optimal parsing problem. Thus, finding an optimal parsing in linear time corresponds to discovering a strategy for using only a subgraph of linear size.

Indeed, in order to get over the quadratic worst case problem, there are many different approaches and many papers deal with optimal parsing in dictionary compressions. For instance the reader can see [1, 2, 5, 7, 9, 10, 12, 13, 17, 19, 22, 24]. Among them, we stress [5] where it is shown that a minimal path can be obtained by using a subgraph of $G_{\mathcal{A},T}$ of size $O(n\ log(n))$, in the LZ77 case under some natural assumptions on the cost function, by exploiting the discreteness of the cost functions.

In this paper we use a similar strategy, i.e. we consider static or dynamical dictionaries, following the approach of [20] and we discover a "small" subgraph of $G_{\mathcal{A},T}$ that is linear in the size of the text for LZ78-alike dictionaries and $O(n \log(n))$ for LZ77-alike dictionaries. This "small" subgraph is such that any minimal path in it is also a minimal path in $G_{\mathcal{A},T}$. Our algorithm has therefore two advantages with respect to the classical Flexible Parsing. First, it can handle *variable cost* of dictionary pointers. This fact allows to extend the range of application of Flexible Parsing to almost all LZ78-alike known algorithms of our extension. Secondly, our Dictionary-Symbolwise Flexible Parsing implemented in the case of LZ77 dictionary gives as particular case when the symbolwise is not in use, a result that is similar to the one presented in [5] that has $O(n\ log(n))$ complexity, using a completely different and simpler subgraph and a simpler data structure. Last but not least our algorithm allows to couple classical LZ-alike algorithms with several symbolwise algorithms to obtain dictionary-symbolwise algorithms that achieve better compression with prove of optimality.

In Section 2 we recall some literature notions about dictionary and dictionary-symbolwise compression algorithms and we define the graph $G_{\mathcal{A},T}$. In Section 3 we formalize the definition of optimal algorithm and optimal parsing and extend them to the dictionary-symbolwise domain. In Section 4 we present the *Dictionary-Symbolwise Flexible Parsing*, a parsing algorithm that extends in some sense the *Flexible Parsing* (*cf.* [19]). We prove its optimality by showing that it corresponds to a shortest path in the full graph, and in Section 5 we describe some data structures that can be used for our algorithm in the two main cases of LZ78-alike and LZ77-alike dictionaries together the time analysis. In Section 6 we state a theoretical result that shows that dictionary-symbolwise compressors can be asymptotically better than dictionary-alone compressors when the dictionary is LZ78 based or when it is LZ77 based. In Section 7 we present some experimental results that show the effectiveness of the dictionary-symbolwise approach. Finally, Section 8 reports our conclusions.

2 Preliminaries

In [1] it is possible to find a survey on Dictionary methods and Symbolwise methods and a description of deep relationships among them (see also [23, 4]).

A dictionary compression algorithm, as noticed in [1], can be fully described by:

1. The dictionary description, i.e. a static collection of phrases or a complete algorithmic description on how the dynamic dictionary is built and updated.
2. The encoding of dictionary pointers in the compressed data.
3. The parsing method, i.e. the algorithm that splits the uncompressed data in dictionary phrases.

We notice that any of the above 3 specifics can all depend on each other, i.e. they can be mutually interdependent.

It is possible to associate a directed weighted graph $G_{\mathcal{A},T} = (V, E, L)$ to any compression algorithm $\mathcal{A}$, any text $T = a_1 a_2 a_3 \cdots a_n$ and any cost function $C : E \to \mathbb{R}^+$ in the following way.

The set of vertices is $V = \{0, 1, \ldots, n\}$, where vertex i corresponds to a_i, i.e. the i-th character in the text T, for $1 \leq i \leq n$ and vertex 0 correspond to the position at the beginning of the text, before any characters. The empty word ϵ is associated to vertex 0 that is also called the *origin* of the graph.

The set of directed edges is $E = \{(p, q) \subset (V \times V) \mid p < q \text{ and } \exists\ w = T[p+1 : q] \in D_p\}$, where $T[p+1 : q] = a_{p+1} a_{p+2} \cdots a_q$ and D_p is the dictionary relative to the processing step p-th, i.e. the step in which the algorithm either has processed the input text up to character a_p, for $0 < p$, or it has to begin, for $p = 0$. For each edge (p, q) in E, we say that (p, q) is *associated to* the dictionary phrase $w = T[p+1 : q] = a_{p+1} \cdots a_q \in D_p$. In the case of static dictionary D_i is constant along the algorithm steps, i.e. $D_i = D_j, \forall i, j = 0 \cdots n$.

L is the set of edge labels $L_{p,q}$ for every edge $(p, q) \in E$, where the label $L_{p,q}$ is defined as the cost of the edge (p, q) when the dictionary D_p is in use, i.e. $L_{p,q} = C((p, q))$. Practically speaking, the cost of an edge is usually set to be the length in bits of its representation in the output, i.e. the selected cost function associate to each edge a cost equal to the length of the encoded dictionary pointer that points out to the phrase in the dictionary that the edge is associated to. Let us notice that the cost of encoding pointers should be a total function, in order to have $L_{p,q}$ always defined for each edge of the graph. There are cases where the cost function is a partial function, i.e. $L_{p,q}$ is not defined for some p and q, and $G_{\mathcal{A},T}$ in such cases is not defined. In this case, one can assign special values, for instance $+\infty$, to labels of edges for which cost function is undefined in order to build an usable graph. When $L_{p,q}$ is always defined for each edge of the graph we say that $G_{\mathcal{A},T}$ is *well defined*. Graph $G_{\mathcal{A},T}$ is not allways *well defined* and we refer to [20] for further discussions on this subject.

Let us remember that the cost function can depend on the parsing if the encoding of dictionary pointers does. For example dictionary pointers could be encoded with Huffman code, as gzip does, and two pointers to the same phrase could have different costs at different time.

Given a compression algorithm $\mathcal{A}$, we call *cost of the encoded text* the sum of all the edges labels in the path from 0 to n relative to the parsing of the algorithm.

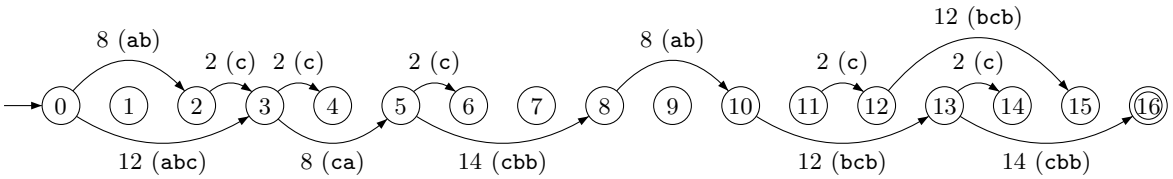

Fig. 1. Graph $G_{\mathcal{A},T}$ for the text T = `abccacbbabbcbcbb`, for the dictionary algorithm $\mathcal{A}$ with static dictionary D = {`ab`, `cbb`, `ca`, `bcb`, `abc`, `c`} and the cost function C as defined in the graph. The dictionary phrase associated to an edge is reported near the edge label within parenthesis.

Let a *scheme of dictionary algorithms* be a set of algorithms that share the first two specifics, i.e. (1) the dictionary description and (2) the encoding of dictionary pointers. A scheme does not necessarily contain *all* algorithms sharing the first two specifics. A parsing algorithm univocally identify a dictionary compression algorithm within a scheme, whenever it belongs to the scheme. Let us notice that the word *scheme* has been used by other authors with other related meaning.

The theory of Dictionary-Symbolwise compression algorithms started in [20] improve and generalize previous results stated for dictionary algorithms and allows to obtain new ones. In what follow, we recall the fundamental notions of the Dictionary-Symbolwise theory.

A *dictionary-symbolwise* algorithm uses both dictionary and symbolwise compression. Such compressors parse the text as a free mixture of dictionary phrases and literal characters, which are substituted by the corresponding pointers or literal codes, respectively. Therefore, the description of a dictionary-symbolwise algorithm should also include the so called *flag information*, that is the technique used to distinguish the actual compression method (dictionary or symbolwise) used for each segment or factor of the parsed text. Often, as in the case of LZSS (*cf.* [22]), an extra bit is added either to each pointer or encoded character to distinguish between them. Encoded information flag can require less (or more) space than one bit.

For instance, a dictionary-symbolwise compression algorithm with a fixed dictionary $D = \{ab, cbb, ca, bcb, abc\}$ and the static symbolwise codeword assignment $[a = 1, b = 2, c = 3]$ could compress the text $abccacbbabbcbcbb$ as $F_d1F_s3F_d3F_d2F_d1F_d4F_d2$, where F_d is the information flag for dictionary pointers and F_s is the information flag for the symbolwise code.

More formally, a parsing of a text T in a dictionary-symbolwise algorithm is a pair $(parse, Fl)$ where $parse$ is a sequence $(u_1, \cdots, u_s)$ of words such that $T = u_1 \cdots u_s$ and where Fl is a boolean function that, for $i = 1, \ldots, s$ indicates whether the word u_i has to be coded as a dictionary pointer or as a symbol. See Table 1 for an example of dictionary-symbolwise compression.

A dictionary-symbolwise compression algorithm is specified by:

1. The dictionary description.
2. The encoding of dictionary pointers.
3. The symbolwise encoding method.
4. The encoding of the flag information.
5. The parsing method.

We can naturally extend the definition of the graph associated to an algorithm for the dictionary-symbolwise case. Given a dictionary-symbolwise algorithm $\mathcal{A}$, a text T and a cost function C defined

Input	ab	c	ca	cbb	ab	bcb	cbb
Output	F_d1	F_s3	F_d3	F_d2	F_d1	F_d4	F_d2

Table 1. Example of compression for the text $abccacbbabbcbcbb$ by a simple Dyctionary-Symbolwise algorithm that use $D = \{ab, cbb, ca, bcb, abc\}$ as static dictionary, the identity as dictionary encoding and the mapping $[a = 1, b = 2, c = 3]$ as symbolwise encoding.

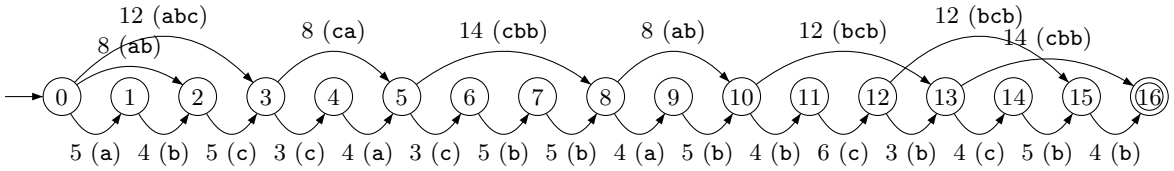

Fig. 2. Graph $G_{\mathcal{A},T}$ for the text $T = \texttt{abccacbbabbcbcbb}$, for the dictionary-symbolwise algorithm $\mathcal{A}$ with static dictionary $D = \{\texttt{ab}, \texttt{cbb}, \texttt{ca}, \texttt{bcb}, \texttt{abc}, \texttt{c}\}$ and cost function C as defined in the graph. The dictionary phrase or the symbol associated to an edge is reported near the edge label within parenthesis.

on edges, the graph $G_{\mathcal{A},T} = (V, E, L)$ is defined as follows. The vertexes set is $V = \{0 \cdots n\}$, with $n = |T|$. The set of directed edges $E = E_d \bigcup E_s$, where $E_d = \{(p, q) \subset (V \times V) \mid p < q - 1, \text{ and } \exists w = T[p+1 : q] \in D_p\}$ is the set of dictionary edges and $E_s = \{(q-1, q) \mid 0 < q \leq n\}$ is the set of symbolwise edges. L is the set of edge labels $L_{p,q}$ for every edge $(p, q) \in E$, where the label $L_{p,q} = C((p, q))$. Let us notice that the cost function C hereby used has to include the cost of the flag information to each edge, i.e. either $C(p, q)$ is equal to $\langle$ the cost of the encoding of $F_d\rangle + \langle$ the cost of the encoded dictionary phrase $w \in D_p$ associated to the edge $(p, q)\rangle$ if $(p, q) \in E_d$ or $C(p, q)$ is equal to (the cost of encoded F_s) + (the cost of the encoded symbol a_q) if $(p, q) \in E_s$. Moreover, since E_d does not contain edges of length one by definition, $G_{\mathcal{A},T} = (V, E, L)$ is not a multigraph. Since this graph approach can be extended to multigraph, with a overhead of formalism, one can relax the $p < q - 1$ constrain in the definition of E_d to $p \leq q - 1$. All the results we will state in this paper, naturally extend to the multigraph case.

We call *dictionary-symbolwise scheme* a set of algorithms having in common the same first four specifics (i.e. they differ one each other for just the parsing methods). A scheme does not need to contain *all* algorithms having the same first four specifics. We notice that any of the specifics from 1 to 5 above can depend on all others, i.e. they can be mutually interdependent. Fixed a *dictionary-symbolwise scheme*, whenever the specifics of the parsing method are given, exactly one algorithm is completely described. Notice that the word *scheme* has been used by other authors with other related meaning. For us the meaning is rigorous.

3 On Optimality

In this section we recall some background notions for the reader to understand the algorithms given in the rest of the paper. However, we assume that the reader is familiar with LZ alike dictionary encoding and with some simple statistical encodings such as the Huffman encoding.

Definition 1. *Fixed a dictionary description, a cost function C and a text T, a* dictionary (dictionary-symbolwise) algorithm *is* optimal *within a set of algorithms if the cost of the encoded text is minimal with respect to all others algorithms in the same set. The parsing of an* optimal *algorithm is called* optimal *within the same set.*

When the length in bit of the encoded dictionary pointers is used as cost function, the previous definition of optimality is equivalent to the classical well known definition of bit-optimality for dictionary algorithm.

Notice that the above definition of optimality strictly depends on the text T and on a set of algorithms. A parsing can be optimal for a text and not for another one. Clearly, we are mainly interested on parsings that are optimal either for *all* texts over an alphabet or for classes of texts. Whenever it is not explicitly written, from now on when we talk about optimal parsing we mean optimal parsing for *all* texts. About the set of algorithm it make sense to find sets as large as possible.

Classically, there is a bijective correspondence between parsings and paths in $G_{\mathcal{A},T}$ from vertex 0 to vertex n, where optimal parses correspond to minimal paths and vice-versa. We say that a parse

(resp. path) *induces* a path (resp. parse) to denote this correspondence. This correspondence was firstly stated in [21] only in the case of sets of algorithms sharing the same *static* dictionary and where the encoding of pointers has constant cost. For example, in Figure 1 we have that the path along vertexes $(0, 2, 3, 5, 8, 10, 13, 16)$ is minimal and it correspond to the optimal parsing of the text for the scheme represented by the graph. While the path along vertexes $(0, 3, 4, 5, 6, 7, 8, 10, 13, 14, 15, 16)$ is the shortest path for the graph in Figure 2. Authors of [20] were the firsts to formally extend the Shortest Path approach to dynamically changing dictionaries and variable costs.

Since graph $G_{\mathcal{A},T}$ are DAGs (Directed Acyclic Graph) that are naturally topologically ordered, classical shortest path algorithms linear in the size of the graph are known. Unfortunately the size of the graph can be quadratic in the size of the text and this approach was not recommended in [21] because it is too time consuming. In the case of LZ78 alike algorithms it is not difficult to show that the size of this graph can be $O(n^{\frac{3}{2}})$, where n is the size of the text.

In order to get over the quadratic (or the $O(n^{\frac{3}{2}})$) worst case problem, there are many different approaches to limit the size of the graph, as already pointed out in the introductive section.

Linear time optimal parsing solution are known for some particular cases. Indeed, under the constant cost assumption, it is proved in [2] the optimality of the *greedy parsing* for suffix-closed static dictionary. In [18, 19, 17] Flexible Parsing is described and it is given a linear time implementation. It is proved that Flexible Parsing is optimal within the set of all algorithms that have the same dictionary of LZW with constant-cost pointers encoding and some experimental results are also described.

The theory of Dictionary-Symbolwise compression algorithms started in [20] improve and generalize previous results stated for dictionary algorithms and allows to obtain new ones.

Let us give the following definition on same class of schemes:

Definition 2. *A scheme $\mathcal{S}$ has the* Schuegraf property *if, for any text T and for any pair of algorithms $\mathcal{A}, \mathcal{A}' \in \mathcal{S}$, the graph $G_{\mathcal{A},T} = G_{\mathcal{A}',T}$ with $G_{\mathcal{A},T}$ well defined.*

This property of schemes is called *property of Schuegraf* in honor to the first of the authors in [21]. In this case we define $G_{\mathcal{S},T} = G_{\mathcal{A},T}$ as the graph of (any algorithm of) the scheme. There is a bijective correspondence between optimal parsings and shortest paths in $G_{\mathcal{S},T}$ from vertex 0 to vertex n.

Definition 3. *Let us consider an algorithm $\mathcal{A}$ and a text T and suppose that graph $G_{\mathcal{A},T}$ is well defined. We say that $\mathcal{A}$ is* dictionary *optimal (with respect to T) if its parsing induces a shortest path in $G_{\mathcal{A},T}$ from the origin to vertex n, with $n = |T|$. In this case we say that its parsing is* dictionary *optimal.*

Let $\mathcal{A}$ be an algorithm such that for any text T the graph $G_{\mathcal{A},T}$ is well defined. We want to associate to it a scheme $\mathcal{SC}_A$ in the following way. Let S be the set of all algorithms $\mathcal{A}$ such that for any text T $G_{\mathcal{A},T}$ exists (i.e. it is well defined). Let $\mathcal{B}$ and $\mathcal{C}$ two algorithms in S. We say that $\mathcal{B}$ and $\mathcal{C}$ are equivalent or $\mathcal{B} \equiv \mathcal{C}$ if for any text T $G_{\mathcal{B},T} = G_{\mathcal{C},T}$. We define the scheme $\mathcal{SC}_A$ to be the equivalence class that has $\mathcal{A}$ as a representative. It is easy to prove that $\mathcal{SC}_A$ has the Schuegraf property.

We can connect the definition of *dictionary* optimal parsing with the previous definition of $\mathcal{SC}_A$ to obtain the next proposition, that says, roughly speaking, that dictionary optimality implies scheme (or global) optimality within the scheme $\mathcal{SC}_A$.

Proposition 1. *Let us consider an algorithm $\mathcal{A}$ such that for any text T the graph $G_{\mathcal{A},T}$ is well defined. Suppose further that for a text T the parsing of $\mathcal{A}$ is dictionary optimal. Then the parsing of $\mathcal{A}$ of the text T is (globally) optimal within the scheme $\mathcal{SC}_A$.*

We have simple examples where a parsing of a text is dictionary optimal and the corresponding algorithm belongs to a scheme that has not the Schuegraf property and it is not (globally) optimal within the same scheme. For pure dictionary scheme having the Schuegraf Property we mean a dictionary-symbolwise scheme having the Schuegraf Property where all algorithms in the scheme are

pure dictionary. We have to be a bit careful using this terminology. Indeed, LZ78, LZW, LZ77 and related algorithms often parse the text with a dictionary pointer and then add a symbol, i.e. the parse phrase is composed by a dictionary pointer and a symbol. In these cases all edges of $G_{\mathcal{A},T}$ denote parse phrases coupled to the corresponding symbol. Edges are labeled by the cost of the dictionary pointer plus the cost of the symbol. We consider these cases included in the class of "pure dictionary" algorithms and schemes.

It is possible to obtain a new statement of the main result concerning flexible parsing revisited in our formalism as follows. The proof is omitted since it is implicitly contained in [19, 17]. It holds for all dictionary algorithms, under the assumptions that the dictionary is at any moment prefix closed and that the cost of encoding pointers is constant. The reader can refer to [19, 17] for further notations and definitions.

Theorem 1. *Flexible Parsing is dictionary optimal.*

4 Dictionary-Symbolwise Flexible Parsing Algorithm

In this section we extend the notion of flexible parsing to the dictionary-symbolwise case and we prove that it is still optimal within any scheme having the Schuegraf Property. We assume here that the dictionary must be at any moment prefix closed. The algorithm is quite different from the original Flexible Parsing but it has some analogies with it and, in the case of LZ78-alike dictionaries, it makes use of one of the main data structures used for the original flexible parsing in order to be implemented in linear time.

Concerning the costs of encoding pointers, we recall that costs can vary but that they assume positive values and that they include the cost of flag information.

Concerning the symbolwise compressor, the costs of symbols must be positive, including the flag information cost. They can vary depending on the position of the character in the text and on the symbol itself.

We suppose further that a text T of length n is fixed and that we are considering the graph $G_{\mathcal{A},T}$, where $\mathcal{A}$ is a dictionary-symbolwise algorithm, and $G_{\mathcal{A},T}$ is well defined under our assumption. We denote by d the function that represent the distance of the vertexes of $G_{\mathcal{A},T}$ from the origin of the graph. Such a distance $d(i)$ is classically defined as the minimal cost of all possible weighted paths from the origin to the vertex i, where $d(0) = 0$. This distance obviously depends on the cost function. We say that cost function C is *prefix-non-decreasing* at any moment if for any $u, v \in D_p$ phrases associated to edges $(p, i), (p, q)$, with $p < i < q$, that implies that u is prefix of v, one has that $C((p, i)) \leq C((p, q))$.

Lemma 1. *Let $\mathcal{A}$ be a dictionary-symbolwise algorithm such that for any text T the graph $G_{\mathcal{A},T}$ is well defined. If the dictionary is always (at any moment) prefix-closed and if the cost function is always (at any moment) prefix-non-decreasing then the function d is non-decreasing monotone.*

Proof. It is sufficient to prove that for any i, $0 \leq i < n$ one has that $d(i) \leq d(i+1)$. Let $j \leq i$ be a vertex such that $(j, i+1)$ is an edge of the graph and $d(i+1) = d(j)+C((j, i+1))$. If j is equal to i then $d(i+1) = d(i)+C((i, i+1))$ and the thesis follows. If j is smaller than i then, since the dictionary D_j is prefix closed, (j, i) is still an edge in D_j and $d(i) \leq d(j)+C((j, i)) \leq d(j)+C((j, i+1)) = d(i+1)$ and the thesis follows. The last inequality in previous equation depend on the fact that the cost function is prefix-non-decreasing.

In what follows in this paper we suppose that the graph $G_{\mathcal{A},T}$ is well defined.

Let us call vertex j a *predecessor* of vertex $i \iff \exists (j, i) \in E$ such that $d(i) = d(j)+C((j, i))$. Let us define $pre(i)$ to be the smallest of the predecessors of vertex $i, 0 < i \leq n$, that is $pre(i) = \min\{j \mid d(i) = d(j)+C((j, i))\}$. In other words $pre(i)$ is the smallest vertex j that contributes to the definition of $d(i)$. Clearly $pre(i)$ has distance smaller than $d(i)$. Moreover the function pre is not necessarily

injective. For instance, a vertex can be a predecessor either "via" a dictionary edge or "via" a symbol edge.

It is also possible to extend previous definition to pointers having a cost smaller than or equal to a fixed c.

Definition 4. *For any cost c we define $pre_c(i) = \min\{j \mid d(i) = d(j) + C((j,i)) \text{ and } C((j,i)) \leq c\}$. If none of the predecessor j of i is such that $C((j,i)) \leq c$ then $pre_c(i)$ is undefined.*

If all costs of the pointers are smaller than or equal to c then for any i one has obviously that $pre_c(i)$ is equal to $pre(i)$.

Analogously to the notation of [17], we want to define two boolean operations *Weighted-Extend* and *Weighted-Exist*.

Definition 5. *Given an edge (i,j) in $G_{\mathcal{A},T}$ and its associated phrase w, a cost value c and a character 'a', the operation* Weighted-Extend$((i,j),a,c)$ *finds out whether the word wa is a phrase in D_i having cost smaller than or equal to c.*

More formally, let (i,j) be such that $w = T[i+1:j] \in D_i$ and, then, (i,j) is in $G_{\mathcal{A},T}$. *Weighted-Extend*$((i,j),a,c)$ = *"yes"* $\iff$ $wa = T[i+1:j+1] \in D_i$ *and* $C((i,j+1)) \leq c$, where C is the cost function associated to the algorithm $\mathcal{A}$. Otherwise *Weighted-Extend*$((i,j),a,c)$ = *"no"*.

Definition 6. *Given $0 < i,j \leq n$ and a cost value c, the operation* Weighted-Exist(i,j,c) *finds out whether or not the phrase $w = T[i+1:j]$ is in D_i and the cost of the corresponding edge (i,j) is smaller than or equal to c.*

Let us notice that doing successfully a *Weighted-Extend* operation on $((i,j),a,c)$ means that $wa \in D_i$ is the weighted extension of w and the encoding of $(i,j+1)$ has cost less or equal to c. Similarly, doing a *Weighted-Exist* operation on (i,j,c) means that an edge (i,j) exist in $G_{\mathcal{A},T}$ having cost less or equal to c.

Let E_c be the subset of all edges of the graph having cost smaller than or equal to c.

Definition 7. *Let us also define, for any cost c the set $M_c \subseteq E_c$ to be the set of* c-supermaximal *edges, where $(i,j) \in M_c \iff (i,j) \in E_c$ and $\forall p,q \in V$, with $p < i$ and $j < q$, the arcs $(p,j),(i,q)$ are not in E_c. For any $(i,j) \in M_c$ let us call i a c-starting point and j a c-ending point.*

Proposition 2. *Suppose that (i,j) and (i',j') are in M_c. One has that $i < i'$ if and only if $j < j'$.*

Proof. Suppose that $i < i'$ and that $j \geq j'$. By the fact that the dictionary D_i is prefix closed we have that (i,j') is still in D_i and therefore it is an edge of $G_{\mathcal{A},T}$. By the prefix-non-decreasing property of function C we have that $C((i,j')) \leq C((i,j)) = c$, i.e. $(i,j') \in E_c$. This contradicts the fact that (i',j') is in M_c and this proves that if $i < i'$ then $j < j'$. Conversely suppose that $j < j'$ and that $i \geq i'$. If $i > i'$ by previous part of the proof we must have that $j > j'$ that is a contradiction. Therefore $i = i'$. Hence (i,j) and (i,j') both belongs to M_c and they have both cost smaller than or equal to c. This contradicts the fact that (i,j) is in M_c and this proves that if $j < j'$ then $i < i'$.

By previous proposition, if $(i,j) \in M_c$ we can think j as function of i and conversely. Therefore it is possible to represent M_c by using an array $M_c[j]$ such that if (i,j) is in M_c then $M_c[j] = i$ otherwise $M_c[j] = Nil$. Moreover the non-*Nil* values of this array are strictly increasing. The positions j having value different from *Nil* are the ending positions.

We want now describe a simple algorithm that outputs all *c-supermaximal* edges scanning the text left-to-right. We call it *Find Supermaximal*(c). It uses the operations *Weighted-Extend* and *Weighted-Exist*. The algorithm starts with $i = 0, j = 1$ and $w = a_1$.The word w is indeed implicitely defined by the arc (i,j) and therefore it will not appear explicitely in the algorithm. At each step j is increased by one and w is set to w concatenated to $T[j]$. The algorithm executes a series of *Weighted-Extend*

until this operation give a positive answer or the end of the text is reached. After a negative answer of *Weighted-Extend*, the algorithm does a series of *Weighted-Exist* increasing i by one until a positive answer.

The algorithm is stated more formally in the following pseudo code.

```
FIND SUPERMAXIMAL (c)
01.   i ← 0, j ← 1
02.   WHILE j < n
03.   DO
04.       WHILE Weighted-Extend((i, j), a_{j+1}, c) = "yes" AND j < n
05.       DO
06.           j ← j + 1
07.       INSERT (i, j) in M_c, j ← j + 1
08.       DO
09.           i ← i + 1
10.       WHILE Weighted-Exist(i, j, c) = "no" AND i < j
```

We notice that when exiting from cycle of lines $4-6$, the cost of the edge (i,j) could still be strictly smaller than c. The function INSERT simply insert the edge (i,j) in the dynamical set M_c. If we represent M_c by an array as described after Proposition 2, function INSERT sets $M_c[j]$ equal to i. Array $M_c[j]$ was initialized by setting all its entries to *Nil*.

Proposition 3. *Above algorithm correctly computes M_c*

Proof. First of all let us prove that if $(\hat{i},\hat{j})$ is inserted by the algorithm in M_c then $(\hat{i},\hat{j})$ is *c-supermaximal*. First of all, since $C((\hat{i},\hat{j})) \leq c$ then the first part of the definition is verified.
Since *Weighted-Extend*$((\hat{i},\hat{j}), a_{\hat{j}+1}, c)$ = "no" then $(\hat{i},\hat{j}+1) \notin E_c$. Since D_i is prefix closed and the function cost C is prefix-non decreasing $\forall q \in V$ with $\hat{j} < q$ the arc $(\hat{i},q)$ is not in E_c for otherwise $(\hat{i},\hat{j}+1)$ would be in E_c.

It remains to prove that $\forall p \in V$ with $p < \hat{i}$ the arc $(p,\hat{j})$ is not in E_c.

Suppose by contradiction that there exists one such arc $(p,\hat{j})$ in E_c.

If $p = 1$, since D_1 is prefix closed and the function cost is prefix-non-decreasing then variable j must reach a value greater than or equal to $\hat{j}+1$ after the first round of *Weighted-Extend* and no further operations can lead to INSERT the arc $(\hat{i},\hat{j})$. This leads to a contradiction.

If $p > 1$ then p has been reached by the variable i after a round of *Weighted-Exist* operations described in lines $8-10$. Let j_p be the corresponding value that the variable j assumes when $i = p$. If $j_p > \hat{j}$ then no further operations can lead to INSERT the arc $(\hat{i},\hat{j})$. A contradiction. If $j_p \leq \hat{j}$, since $(p,\hat{j})$ in E_c and the dictionary D_p is prefix closed and the function cost is prefix-non-decreasing we have that for any j with $j_p \leq j \leq \hat{j}$ the arc $(p,j) \in E_c$. Therefore when variable i reaches p it must start a round of *Weighted-Extend* until j assume a value greater than or equal to $\hat{j}+1$. Even in this case no further operations can lead to INSERT the arc $(\hat{i},\hat{j})$. This leads to a contradiction. Therefore if $(\hat{i},\hat{j})$ is inserted by the algorithm in M_c then $(\hat{i},\hat{j})$ is *c-supermaximal*.

We have now to prove that if $(\hat{i},\hat{j})$ is *c-supermaximal* then it is inserted by the algorithm in M_c.

Suppose that variable i never assumes the value $\hat{i}$. The algorithm ends when variable j is equal to n. Let i_n be the value of variable i just before that j becomes n. Since variable j increases only in line 6 after that the operation *Weighted-Extend* outputs "yes", we know that $(i_n,n) \in E_c$. Since The dictionary D_{i_n} is prefix closed and $\hat{j} \leq n$ and since the function cost is prefix-non-decreasing then $(i_n,\hat{j}) \in E_c$. Since variable i ranged from 1 up to i_n then $i_n < \hat{i}$ and this contradicts the fact that $(\hat{i},\hat{j})$ is *c-supermaximal*.

Therefore at a certain moment variable i is assuming the value $\hat{i}$. Let $j_{\hat{i}}$ be the value of variable j in that moment. Variable i increases only during a round of operations *Weighted-Exist*. Before this

round there must have been a round of operations *Weighted-Extend*. Let us consider the last one of this round. Variable i is assuming the value $i' < \hat{i}$ and, before increasing the variable j we know that $(i', j_{\hat{i}} - 1)$ is in E_c.

If, at this point, $j_{\hat{i}} > \hat{j}$ then $j_{\hat{i}} - 1 \geq \hat{j}$. Since the dictionary $D_{i'}$ is prefix closed and since the function cost is prefix-non-decreasing, $(i', \hat{j}) \in E_c$. This contradicts the fact that $(\hat{i}, \hat{j})$ is *c-supermaximal*. Hence $j_{\hat{i}} \leq \hat{j}$.

Since $(\hat{i}, \hat{j})$ is in E_c, again since the dictionary $D_{\hat{i}}$ is prefix closed and since the function cost is prefix-non-decreasing, for all j, $j_{\hat{i}} \leq j \leq \hat{j}$ we have that $(\hat{i}, j) \in E_c$. Therefore when variable i reaches $\hat{i}$ it starts a sequence of *Weighted-Extend* and the variable j reaches the value $\hat{j}$. It and cannot go further for otherwise $(\hat{i}, \hat{j})$ would not be *c-supermaximal*. Therefore $(\hat{i}, \hat{j})$ is inserted in M_c.

Proposition 4. *For any edge $(i, j) \in E_c$ there exists a* c-supermaximal *edge $(\hat{i}, \hat{j})$ containing it, e.g. such that $\hat{i} \leq i$ and $j \leq \hat{j}$.*

Proof. We build $(\hat{i}, \hat{j})$ in algorithmic fashion. The algorithm is described in what follows in an informal but rigorous way. If edge (i, j) is not *c-supermaximal* then we proceed with a round of *Weighted-Extend*$((i, j), a_{j+1}, c)$ analogously as described in algorithm FIND SUPERMAXIMAL and increase j of one unit until *Weighted-Extend* outputs "no". Let j_1 be last value of j for which *Weighted-Extend* output "yes". Clearly $(i, j_1) \in E_c$ and $(i, j) \neq (i, j_1)$. If (i, j_1) is not *c-supermaximal* the only possibility is that there exists at least one $i_1 < i$ such that $(i_1, j_1) \in E_c$. At this point we keep iterating previous two steps starting from (i_1, j_1) instead of (i, j) and we stops whenever we get a *c-supermaximal* edge, that we call $(\hat{i}, \hat{j})$.

By previous proposition for any node $v \in G_{\mathcal{A},T}$ if there exists a node $i < v$ such that $C((i, v)) = c$ and $d(v) = d(i) + c$ then there exists a *c-supermaximal* edge $(\hat{i}, \hat{j})$ containing (i, v) and such that $\hat{j}$ is the *closest* arrival point greater that v. Let us call this *c-supermaximal* edge $(\hat{i}_v, \hat{j}_v)$. We use $\hat{i}_v$ in next proposition.

Proposition 5. *Suppose that $v \in G_{\mathcal{A},T}$ is such that there exists a previous node i such that $C((i, v)) = c$ and $d(v) = d(i) + c$. Then $\hat{i}_v$ is a predecessor of v, e.g. $d(v) = d(\hat{i}_v) + C((\hat{i}_v, v))$ and, moreover, $d(\hat{i}_v) = d(i)$ and $C((\hat{i}_v, v)) = c$.*

Proof. Since $(\hat{i}_v, \hat{j}_v)$ contains (i, v) and the dictionary at position $\hat{i}_v$ is prefix close then $(\hat{i}_v, v)$ is an edge of $G_{\mathcal{A},T}$. Since $(\hat{i}_v, \hat{j}_v)$ has cost smaller than or equal to c then, by the suffix-non-decreasing property, also $(\hat{i}_v, v)$ has cost smaller than or equal to c. Since the distance d is non-decreasing we know that $d(\hat{i}_v) \leq d(i)$. By very definition of the distance d we know that $d(v) \leq d(\hat{i}_v) + C((\hat{i}_v, v))$.

Putting all together we have that $d(v) \leq d(\hat{i}_v) + C((\hat{i}_v, v)) \leq d(i) + c = d(v)$. Hence the inequalities in previous equation must be equalities and, further, $d(\hat{i}_v) = d(i)$ and $C((\hat{i}_v, v)) = c$.

Corollary 1. *For any vertex v, the edge $(\hat{i}_v, v)$ is the last edge of a path of minimal cost from the origin to vertex v.*

Proof. Any edge x in $G_{\mathcal{A},T}$ that is such that $d(v) = d(x) + C((x, v))$ is the last edge of a path of minimal cost from the origin to vertex v.

In what follows we describe a graph $G'_{\mathcal{A},T}$ that is a subgraph of $G_{\mathcal{A},T}$ that is such that for any node $v \in G_{\mathcal{A},T}$ there exists a minimal path from the origin to v in $G'_{\mathcal{A},T}$ that is also a minimal path from the origin to v in $G_{\mathcal{A},T}$. The proof of this property, that will be stated in the subsequent proposition, is a consequence of Proposition 5 and Corollary 1.

We describe the building of $G'_{\mathcal{A},T}$ in an algorithmic way. Even if we do not give the pseudocode, algorithm BUILD $G'_{\mathcal{A},T}$ is described in a rigorous way and it makes use, as a part of it, of algorithm FIND SUPERMAXIMAL.

The set of nodes of $G'_{\mathcal{A},T}$ is the same of $G_{\mathcal{A},T}$. First of all we insert all symbolwise edges of $G_{\mathcal{A},T}$ in $G'_{\mathcal{A},T}$. Let now $\mathcal{C}$ be the set of all possible costs that any dictionary edge has. This set can be build starting from $G_{\mathcal{A},T}$ but in all known meaningful situations the set $\mathcal{C}$ is usually well known and can be ordered and stored in an array in a time that is linear in the size of the text.

For any $c \in \mathcal{C}$ we use algorithm FIND SUPERMAXIMAL to obtain the array $M_c[j]$. For any *c-supermaximal* edge (i, j), we add in $G'_{\mathcal{A},T}$ all edges of the form (i, x) where x varies from j down to (and not including) the previous arrival position j' if this position is greater than $i+1$ otherwise down to $i+2$. More formally, for any j such that $M_c[j] \neq Nil$ let j' be the greatest number smaller than j such that $M_c[j] \neq Nil$. For any x, such that $\max(j', i+2) \leq x \leq j$, add (i, x) to $G'_{\mathcal{A},T}$ together with its label. This concludes the construction of $G'_{\mathcal{A},T}$.

Since (i, j) and the dictionary D_i is prefix closed then all previous arcs of the form (i, x) are also arcs of $G_{\mathcal{A},T}$ and, therefore, $G'_{\mathcal{A},T}$ is a subgraph of $G_{\mathcal{A},T}$.

Proposition 6. *For any node $v \in G_{\mathcal{A},T}$ there exists a minimal path from the origin to v in $G'_{\mathcal{A},T}$ that is also a minimal path from the origin to v in $G_{\mathcal{A},T}$.*

Proof. The proof is by induction on v. If v is the origin there is nothing to prove. Suppose now that v is greater than the origin and let (i, v) the last edge of a minimal path in $G_{\mathcal{A},T}$ from the origin to v. By inductive hypothesis there exists a minimal path $\mathcal{P}$ from the origin to i in $G'_{\mathcal{A},T}$ that is also a minimal path from the origin to i in $G_{\mathcal{A},T}$. Since (i, v) is a symbolwise arc then it is also in $G'_{\mathcal{A},T}$ and the concatenation of above minimal path $\mathcal{P}$ with (i, v) is a minimal path from the origin to v in $G'_{\mathcal{A},T}$ that is also a minimal path from the origin to v in $G_{\mathcal{A},T}$.

Suppose now that (i, v) is a dictionary arc and that its cost is c.

Since it is the last edge of a minimal path we have that $d(v) = d(i) + c$. By Proposition 5 $d(v) = d(\hat{i}_v) + C((\hat{i}_v, v))$ and, moreover, $d(\hat{i}_v) = d(i)$ and $C((\hat{i}_v, v)) = c$.

By Corollary 1, the edge $(\hat{i}_v, v))$ is the last edge of a path of minimal cost from the origin to vertex v. By inductive hypothesis there exists a minimal path $\mathcal{P}$ from the origin to $\hat{i}_v$ in $G'_{\mathcal{A},T}$ that is also a minimal path from the origin to i in $G_{\mathcal{A},T}$. Since $(\hat{i}_v, v))$ has been added by construction in $G'_{\mathcal{A},T}$, the concatenation of above minimal path $\mathcal{P}$ with $(\hat{i}_v, v))$ is a minimal path from the origin to v in $G'_{\mathcal{A},T}$ that is also a minimal path from the origin to v in $G_{\mathcal{A},T}$.

We can now finally describe the *Dictionary-symbolwise flexible parsing*.

The *Dictionary-symbolwise flexible parsing* firstly uses algorithm BUILD $G'_{\mathcal{A},T}$ and then uses the classical SINGLE SOURCE SHORTEST PATH algorithm to recover a minimal path from the origin to the end of graph $G_{\mathcal{A},T}$. The correctness of above algorithm is stated in the following theorem and it follows from above description and from Proposition 6

Theorem 2. Dictionary-symbolwise flexible parsing *is dictionary optimal.*

With respect to the original Flexible Parsing algorithm we gain the fact that it can works with variable costs of pointers and that it is extended to the dictionary-symbolwise case. But we loose the fact that the original one was "on-line". A minimal path has to be recovered, starting from the end of the graph backward. But this is an intrinsic problem that cannot be eliminated. Even if the dictionary edges have just one possible cost, in the dictionary-symbolwise case it is possible that any minimal path for a text T is totally different from any minimal path for the text Ta, that is the previous text T concatenated to the symbol 'a'. Even if the cost of pointers is constant. The same can happen when we have a "pure dictionary" case with variable costs of dictionary pointers. In both cases for this reason, there cannot exists "on-line" optimal parsing algorithms, and, indeed, flexible parsing fails being optimal in the pure dictionary case when costs are variable.

On the other hand our algorithm is suitable when the text is cut in several blocks and, therefore, in practice there is not the need to process the whole text but it suffices to end the current block in order to have the optimal parsing (relative to that block). As another alternative, one can keep track

of just one minimal path all along the text and can use some standard tricks to arrange it if it does not reach the text end, i.e. the wished target node. In the last cases one get a suboptimal solution that is a path with a cost extremely close to the minimal path.

5 Data Structures and Time Analysis

In this subsection we analyze *Dictionary-symbolwise flexible parsing* in both LZ78 and LZ77 alike algorithms.

Concerning LZ78 alike algorithms, the dictionary is prefix closed and it is usually implemented by using a technique that is usually referred as LZW implementation. We do not enter in details of this technique. We just recall that the cost of pointers increases by one unit whenever the dictionary size is "close" to a power of 2. The moment when the cost of pointers increases is clear to both encoder and decoder. In our dictionary-symbolwise setting, we suppose that the flag information for dictionary edges is constant. We assume therefore that it takes $O(1)$ time to determine the cost of all dictionary edges outgoing node i.

The maximal cost that a pointer can assume is smaller than $\log_2(n)$ where n is the text-size. Therefore the set $\mathcal{C}$ of all possible costs of dictionary edges has logarithmic size and it is cheap to calculate.

In [17] it is used a data structure, called trie-reverse pair, that is able to perform the operation of *Extend* and *Contract* in $O(1)$ time.

Since at any position we can calculate in $O(1)$ time the cost of outgoing edges, we can use the same data structure to perform our operations of *Weighted-Extend* and of *Weighted-Exist* in constant time. In order to perform a *Weighted-Extend* we simply use the *Extend* on the same non-weight parameters and, if the answer is "yes" we perform a further check in $O(1)$ time on the cost. In order to perform a *Weighted-Exist* we simply use the *contract* on the same non-weight parameters and, if the answer is "yes" we perform a further check in $O(1)$ time on the cost.

For any cost c finding M_c and the corresponding arcs in order to build ${G'}_{\mathcal{A},T}$ takes then linear time. Therefore, at a first look, performing the algorithm BUILD ${G'}_{\mathcal{A},T}$ would take $O(n\ log(n))$. But, since there is only one cost active at any position then if $c < c'$ then $M_c \subseteq M_{c'}$ as stated in the following proposition.

Proposition 7. *Suppose that for any i the cost of all dictionary pointers in D_i is a constant c_i and that for any i, $0 \leq i < n$ one has that $c_i \leq c_{i+1}$.*

If $c < c'$ then $M_c \subseteq M_{c'}$.

Proof. We have to prove that for any $(i,j) \in M_c$ then $(i,j) \in M_{c'}$. Clearly if $(i,j) \in M_c$ then its cost is smaller than or equal to $c < c'$. It remains to prove that (i,j) is c'-supermaximal, e.g. that $\forall p, q \in V$, with $p < i$ and $j < q$, the arcs $(p,j),(i,q)$ are not in $E_{c'}$. Since $(i,j) \in M_c$ and since the cost of (i,j) is by hypothesis equal to c_i, we have that $c_i \leq c$. If arc (p,j) is in $E_{c'}$ then its cost is $c_p \leq c_i \leq c$ and therefore it is also in E_c contradicting the c-supermaximality of (i,j). If arc (i,q) is in $E_{c'}$ then its cost is $c_i \leq c$ and therefore it is also in E_c contradicting the c-supermaximality of (i,j).

Definition 8. *We say that a cost function C is* LZW-alike *if for any i the cost of all dictionary pointers in D_i is a constant c_i and that for any i, $0 \leq i < n$ one has that $c_i \leq c_{i+1}$.*

At this point, in order to build ${G'}_{\mathcal{A},T}$ it suffices to build M_c where c is the greatest possible cost. Indeed it is useless checking for the cost and one can just use the standard operation *Extend* and *Contract*. Those operation can be implemented in $O(1)$ time using the trie reverse trie data structure for LZ78 standard dictionary or for the LZW dictionary or for the FPA dictionary (cf. [17]). Indeed we call a dictionary *LZ78-alike* if the operations *Extend* and *Contract* can be implemented in $O(1)$ time using the trie reverse trie data structure.

We notice that previous definition of LZ78-alikeness can be relaxed by asking that the operations *Extend* and *Contract* can be implemented in $O(1)$ amortized time using any data structure, including obviously the time used for building such data structure.

The overall time for building $G'_{\mathcal{A},T}$ is therefore linear, as well as its size. The Single Source Shortest Path over $G'_{\mathcal{A},T}$, that is a DAG topologically ordered, takes linear time.

In conclusion we have proved the following theorem.

Theorem 3. *Suppose that we have a dictionary-symbolwise scheme, where the dictionary is LZ78-alike and the cost function is LZW-alike. The symbolwise compressor is supposed to be, as usual, linear time. Using the trie-reverse pair data structure,* Dictionary-Symbolwise flexible parsing *is linear.*

Concerning LZ77, in [5] it has been given, with a similar shortest path approach, an optimal parsing algorithm under some assumptions on the cost function. Our prefix-non-decreasing assumption is weaker than their assumptions in the sense that it is a consequence of their assumptions (*cf.* [5, Fact 4]). The maximal cost that a pointer can have under their assumption is still $O(\log(n))$ where n is the size of the text. It seems that it is possible to use the data structure used in [5] to perform, for any cost, *Weighted-Extend* and *Weighted-Exist* in amortized $O(1)$ time. Then the overall time for the *Dictionary-symbolwise flexible parsing* when the dictionary is LZ77 alike would be $O(n \log(n))$, extending their result to the dictionary-symbolwise case. The subgraph $G'_{\mathcal{A},T}$ of $G_{\mathcal{A},T}$ is totally different from the one used in [5].

Indeed, quite recently, we discovered a simpler data structure that allows us to perform, for any cost, *Weighted-Extend* and *Weighted-Exist* in amortized $O(1)$ time. This data structure is built by using in a clever way $O(\log(n))$ suffix trees and it will be described in the journal version of this paper.

6 Dictionary-Symbolwise Can Have Better Ratio

In this section we prove that there exists a family of strings such that the ratio between the compressed version of the strings obtained by using an optimal LZ78 parsing (with constant cost encoding of pointers) and the compressed version of the strings obtained by using an optimal dictionary-symbolwise parsing is unbounded. The dictionary, in the dictionary-symbolwise compressor is still the LZ78 dictionary, while the symbolwise is a simple Last Longest Match Predictor that will be described later. We want to notice here that similar results were proved in [19] between flexible parsing and the classical LZ78 and in [5] between a compressor that uses optimal parsing over a LZ77 dictionary and the standard LZ77 compressor (*cf.* also [14]). Last but not least we notice that in this example, analogously as done in [19], we use an unbounded alphabet just to make the example clearer. An analogous result can be obtained with a binary alphabet with a more complex example.

Let us define a Dictionary-Symbolwise compressor that uses LZ78 as dictionary method, the Last Longest Match Predictor as symbolwise method, Run Length Encoder to represent the flag information and one optimal parsing method. Let us call it OptDS-LZ78. We could have used a PPM* as symbolwise compressor but Last Longest Match Predictor fits our purposes and it is simple to analyze. Last Longest Match Predictor is just a simple symbolwise compression method that uses the last longest seen match to predict next char.

The symbolwise searches, for any position k of the text, the closest longest block of consecutive letters up to position $k-1$ that is equal to a suffix ending in position k. This compressor predicts the $(k+1)$-th character of the text to be the character that follows the block. It writes a symbol 'y' (that is supposed not to be in the text) if this is the case. Otherwise it uses an escape character 'n' (that is supposed not to be in the text) and then writes down the correct character plainly. A temporary output alphabet has therefore two characters more than the characters in the text. This temporary output will be subsequently encoded by a run-length encoder. This method is like the Yes?No version of Symbol Ranking by P. Fenwick.

Let us consider a string S that is the concatenation of all the prefixes of $1..k$ in increasing order. Let consider the string T' that is the concatenation of the first $\sqrt{k}$ suffixes of $2..k$, i.e. $T' = 2..k \cdot 3..k \cdot \ldots \cdot \sqrt{k}..k$ and a string $T = S \cdot T'$. We use S to build a dictionary formed by just the string $1..k$ and its prefixes and no more. We assume that both the dictionary and the symbolwise methods work the same up to the end of the S string, so they produce an output that is very similar in terms of space. It is not difficult to prove that an optimal LZ78 compressor would produce on T a parse having cost at least $O(k + k\log(k)) = O(k\log(k))$ while the optimal dictionary-symbolwise compressor (under the constant cost assumption on encoding pointers) has a cost that is $O(k + \sqrt{k}\log(k)) = O(k)$.

A similar result can be stated and proved also in the case of LZ77 dictionary.

7 Experimental Results

We present some experimental results on LZ78 alike dictionaries that show the effectiveness of this approach. We did some experiments focused on the gain of an optimal dictionary-symbolwise over an optimal pure dictionary methods, using our flexible version of a LZ78 dictionary-symbolwise compressor, where the dictionary size (number of phrases) is 2^{24}, the symbolwise compressor is an Huffman coding over all symbols and the parse is optimal. We mean that the cost of a symbol is set to be the cost in bits that a static Huffman coding over the whole text assigns to it. We further used an Huffman coding on the flag information by considering 8 consecutive flag information as a character. In order to see the gain over a pure dictionary method, we forced the parsing not to use symbolwise arcs. Therefore, what we call "the pure dictionary method" is essentially the Flexible Parsing of Mathias and Shainalp.

We tested our compressors over some prefixes of the English wikipedia that are widely used as dataset for benchmark, (*cf.* [15]). The prefix of length 10^8 symbols is called ENWIK8. Over ENWIK8 the gain of our dictionary-symbolwise with respect to the pure dictionary method was close to 5%.

We observed a similar gain for other prefixes of English wikipedia, slowly decreasing as the size of the prefix was increasing.

From a formal point of view, for the compression algorithm used by us we cannot build the weighted graph $G_{\mathcal{A},T}$, i.e. the cost of arcs cannot be given "a priori". Nevertheless, in practice we assigned a presumed cost to each arc and we worked as in the case of a compressor that is such that for any text the weighted graph $G_{\mathcal{A},T}$ exists. So, we cannot say that the obtained parse is optimal, but we believe that it is very close to the optimal one. Actually our compressed version of ENWIK8 is less than 90% of the corresponding version compressed by *gzip* with the option -9. Next table shows the percentage of our compressed version w.r.t. gzip-9 ones.

The results described in [17] do not use LZ78 but the LZW with a dictionary of size 2^{16} and the FPA algorithm with a dictionary of size 2^{16} and 2^{24}. This last seems to behave better than LZW and of our LZ78 optimal pure dictionary. Our results are then not directly comparable but they seem to be compatible. Some percentile can be lost or gained by the ability of the programmer. Our program is just a working prototype and its purpose is to compare the gain obtained when an simple Huffman is coupled with a LZ78 with respect to a pure LZ78, both with an optimal parsing. The reader can make other comparisons with other standard compressors such as Bzip2 by using the URL in [15] and gzip -9 as a benchmark.

As further improvement, we gained few percentile points by using a dynamic Huffman coding on the symbols chosen by the parse. We decided to make more "rounds" of compression by assigning the

input file size	$2MB$	$4MB$	$8MB$	$16MB$	$32MB$	$64MB$	$100MB$
pure dictionary	112.97%	109.29%	104.41%	100.88%	98.68%	95.92%	94.28%
dict-symbolwise	107.43%	103.97%	99.37%	96.04%	94.07%	91.51%	89.98%

Table 2. Compression evaluation against $gzip - 9$.

value corresponding to the last distribution to the next round of compression. We noticed an extremely fast stabilization of the values, i.e. just after 3 or at most 4 rounds. We noticed further that different starting distributions could lead to different, even if close, stabilized or stationary distributions, that, in turn, lead to different compressed outputs of similar but not equal size.

As a preliminary result, we observed a much higher gain of a further 10% when the Huffman coding was replaced by an order-1 arithmetical coding. In this case we have just evaluated the cost of the chosen parsing because we have not yet the working compression - decompression program.

Concerning LZ77 dictionary-symbolwise we have implemented a generalisation to larger dictionary size of gzip, that we call Lgzip (Large gzip). When the dictionary size is smaller that or equal to 32 Kb we obtain files that are .gz compliant, i.e. they can be decompressed by usual decompressors. Our compression ratios are almost equal (usually few bytes more indeed) than the equivalent files compressed by deflate64 with maximal compression parameters. These files have by far a much better compression ratio than the one obtained by gzip with the option -9. The library deflate64 is included in the widely used 7zip program.

When we enlarge the dictionary size we obtain similar compression ratio of other commercial compression programs.

When we deal with files of size of 20 megas the advantage of using a symbolwise coupled with a LZ77 dictionary reduces down to a $2-3\%$, less than the 5% of the LZ78 case.

The compression ratio in the LZ77+Huffman case is better than the one in the LZ78+Huffman case. The distance seems to reduce when instead of an Huffmann coding it is used an order-1 arithmetical coding.

We plan to continue these experiments since this direction seems to be very promising. As further experimental work we plan to check our algorithm by using as dictionary some version of FPA algorithm and, eventually, a Rolz compressor.

8 Conclusions

In this paper we present some advancement on dictionary-symbolwise theory. We describe the *Dictionary-Symbolwise Flexible Parsing*, a parsing algorithm that extends in non obvious way the *Flexible Parsing* (*cf.* [19]) to *variable* and *unbounded* costs and to the dictionary-symbolwise algorithm domain. We prove its optimality for prefix-closed dynamic dictionaries under some reasonable assumption. *Dictionary-Symbolwise Flexible Parsing* is *linear* for LZ78 alike dictionaries and even if it is not able to run online it allow to easily make a block programming implementation and a near to optimal online implementation, too. In the case of LZ77 alike dictionary and under lighter constrains on cost function, we have reobtained the $O(n \log(n))$ complexity as authors of [5] recently did and we use a completely different and simpler subgraph and a simpler data structure. We proved that dictionary-symbolwise compressors can be asymptotically better than optimal pure dictionary compression algorithms in compression ratio terms, with LZ78 based dictionary and the same can be proved for LZ77 based dictionary.

Finally, we easily obtained experimentally a 5% enhancement in compression ratio with respect of the pure dictionary compressor building the dictionary-symbolwise LZ78-Huffman compressor. We plan to extend our experimentation on LZ alike dictionary algorithms and many other symbolwise algorithms, since this direction seems to be very promising.

References

1. Timothy C. Bell and Ian H. Witten. The relationship between greedy parsing and symbolwise text compression. *J. ACM*, 41(4):708–724, 1994.
2. Martin Cohn and Roger Khazan. Parsing with prefix and suffix dictionaries. In *Data Compression Conference*, pages 180–189, 1996.
3. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition, 2001.

4. Maxime Crochemore and Thierry Lecroq. Pattern-matching and text-compression algorithms. *ACM Comput. Surv.*, 28(1):39–41, 1996.
5. Paolo Ferragina, Igor Nitto, and Rossano Venturini. On the bit-complexity of lempel-ziv compression. In *SODA '09: Proceedings of the Nineteenth Annual ACM -SIAM Symposium on Discrete Algorithms*, pages 768–777, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.
6. Gzip's Home Page: `http://www.gzip.org`.
7. Alan Hartman and Michael Rodeh. *Optimal parsing of strings*, pages 155–167. Springer - Verlag, 1985.
8. R. N. Horspool. The effect of non-greedy parsing in ziv-lempel compression methods. In *Data Compression Conference*, 1995.
9. Jyrki Katajainen and Timo Raita. An approximation algorithm for space-optimal encoding of a text. *Comput. J.*, 32(3):228–237, 1989.
10. Jyrki Katajainen and Timo Raita. An analysis of the longest match and the greedy heuristics in text encoding. *J. ACM*, 39(2):281–294, 1992.
11. Phil Katz. Pkzip archiving tool, `http://en.wikipedia.org/wiki/pkzip`, 1989.
12. Tae Young Kim and Taejeong Kim. On-line optimal parsing in dictionary-based coding adaptive. *Electronic Letters*, 34(11):1071–1072, 1998.
13. Shmuel T. Klein. Efficient optimal recompression. *Comput. J.*, 40(2/3):117–126, 1997.
14. S. Rao Kosaraju and Giovanni Manzini. Compression of low entropy strings with lempel–ziv algorithms. *SIAM J. Comput.*, 29(3):893–911, 2000.
15. Matt Mahoney. Large text compression benchmark: `http://mattmahoney.net/text/text.html`.
16. Christian Martelock. Rzm order-1 rolz compressor, `http://encode.ru/forums/index.php?action=vthread&forum=1&topic=647`, 4 2008.
17. Yossi Matias, Nasir Rajpoot, and S"uleyman Cenk Sahinalp. The effect of flexible parsing for dynamic dictionary-based data compression. *ACM Journal of Experimental Algorithms*, 6:10, 2001.
18. Yossi Matias and S"uleyman Cenk Sahinalp. On the optimality of parsing in dynamic dictionary based data compression `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.4292`, 1998.
19. Yossi Matias nd S"uleyman Cenk Sahinalp. On the optimality of parsing in dynamic dictionary based data compression. In *SODA*, pages 943–944, 1999.
20. G. Della Penna, A. Langiu, F. Mignosi, and A. Ulisse. Optimal parsing in dictionary-symbolwise data compression schemes. `http://www.di.univaq.it/~mignosi/ulicompressor.php`, 2006.
21. Ernst J. Schuegraf and H. S. Heaps. A comparison of algorithms for data base compression by use of fragments as language elements. *Information Storage and Retrieval*, 10(9-10):309–319, 1974.
22. James A. Storer and Thomas G. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982.
23. John G. Cleary Timothy C. Bell and Ian H. Witten. *Text compression*. Prentice Hall, 1990.
24. Robert A. Wagner. Common phrases and minimum-space text storage. *Commun. ACM*, 16(3):148–152, 1973.